Journal of Discrete Algorithms 9 (2011) 153–165

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Missing pattern discovery

Stanislav Angelov [a], Shunsuke Inenaga [b,*], Teemu Kivioja [c,d], Veli Mäkinen [d]

[a] Department of Computer and Information Science, School of Engineering and Applied Sciences, University of Pennsylvania, Philadelphia, PA 19104, USA
[b] Graduate School of Information Science and Electrical Engineering, Kyushu University, Moto'oka 744, Fukuoka 819-0395, Japan
[c] Genome-Scale Biology Program, Biomedicum Helsinki, P.O. Box 63 (Haartmaninkatu 8), FIN-00014 University of Helsinki, Finland
[d] Department of Computer Science, P.O. Box 68 (Gustaf Hällströmin katu 2b), FIN-00014 University of Helsinki, Finland

ARTICLE INFO

Article history:
Received 8 November 2008
Accepted 2 August 2010
Available online 6 November 2010

Keywords:
Pattern discovery
Polymerase chain reaction (PCR)
Suffix trees

ABSTRACT

In this paper, we study the *missing patterns problem*: Find the shortest *pair of patterns* that do not occur close to each other in a given text, i.e., the distance between their occurrences is always greater than a given threshold α. We present various solutions to this problem, as well as to the case where the patterns in the pair are required to be of the same length. This work is motivated by optimizing the sensitivity of PCR. Experiments show that our algorithm is practical enough to handle human genome data.

1. Introduction

1.1. Biological pattern discovery

Pattern discovery is a fundamental problem in Computational Biology and Bioinformatics [34,4,32,30]. A large amount of effort has been devoted to developing efficient algorithms to extract interesting, useful, and surprising substring patterns (i.e., patterns with no mismatches) from massive biological sequences [31,9]. Then this research has been extended to more advanced pattern classes such as subsequence patterns [7,17], episode patterns [27,18], VLDC patterns [20], and their variations [33]. In particular, finding string patterns of some distinctive characteristic is a central task in knowledge discovery from textual data [4,32]. One extreme example of surprising patterns is *missing patterns*, namely, patterns that do *not* appear in a given text T. Amir et al. [1] introduced a generalized version of the missing pattern problem where the aim is to find a pattern that has the maximum average hamming distance to the text. They call this problem the *inverse pattern matching* problem. Some improvements for this inverse problem were presented in [14]. Another related work is the *farthest substring problem* [24], where a set of text strings is considered as input.

Missing pattern discovery has also recently been popularized[1] as a search for *absent sequences* over all species sequenced so-far [16]; such minimum length absent sequences are argued to be possibly lethal DNA, as they are avoided by evolution.

The demand for *composite pattern discovery* has recently arisen as an extension to the discovery of single patterns. It is motivated by, for instance, the fact that many of the actual regulatory signals are composite patterns that are groups of monad patterns occurring near each other [12]. The concept of composite patterns was introduced by Marsan and Sagot [28] as *structured motifs* which are two or more patterns separated by a certain distance. They presented suffix tree [35] based

* Corresponding author.
E-mail addresses: angelov@cis.upenn.edu (S. Angelov), inenaga@c.csce.kyushu-u.ac.jp (S. Inenaga), teemu.kivioja@helsinki.fi (T. Kivioja), vmakinen@cs.helsinki.fi (V. Mäkinen).

[1] E.g., see the bumper book of DNA no-nos, New Scientist, 6 January 2007.

1570-8667/$ – see front matter
doi:10.1016/j.jda.2010.08.005

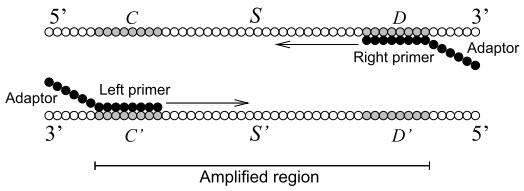

Fig. 1. Polymerase chain reaction (PCR).

algorithms for finding structured motifs and, subsequently, Carvalho et al. [11] gave a new algorithm with improved running time and space.

In a similar concept, Arimura et al. [5,6] introduced *proximity patterns* and proposed algorithms to find these patterns efficiently. MITRA [12] is another method that looks for composite patterns. BioProspector [26] applies the Gibbs sampling strategy to discover gapped motifs. *Boolean combinations* of patterns were considered in [8,19], in order to find regulatory elements that cooperate, complement, or compete with each other in enhancing and/or silencing certain genomic functions.

In this paper, we study a combination of the missing pattern discovery and the composite pattern discovery problems: Given a text T of length n and threshold value α, find the *shortest pair of patterns* such that the distance between their occurrences in T is *always greater than* α. Not only is our missing patterns problem interesting in theory, but it is also well-motivated in practice. An example of numerous potential applications of missing patterns (e.g., see [16]) is to optimize the sensitivity of polymerase chain reaction (PCR). In PCR a pair of short fragments of DNA called primers is specifically designed for the amplified region so that each of them is complementary to the 3′ end of one of two strands of the region (see Fig. 1). The shortest pair of missing patterns for both strands S and S' w.r.t. distance threshold α can be used to design good adaptors for multiplexed PCR primers [21]. Several regions can be further amplified in parallel with one pair of primers complementary to the adaptors if the adaptor sequences have been chosen so that they do not occur close to each other in the sample DNA.

1.2. Summary of results

Firstly, we show that the problem of finding a shortest missing pair is equivalent to the problem of finding a single shortest missing pattern. Then, we introduce a suffix tree [35] based approach to the more general problem of finding all the shortest missing pattern pairs under constraints on the length of each piece. We present an $O(n^2)$-time $O(n)$-space algorithm, and an $O(\alpha n \log n)$-time $O(n \log n)$-space algorithm to solve this problem. Then we develop algorithms based on simple bijective mapping approach. The method solves in $O(\alpha n \log_\sigma n)$ time the general problem and in $O(\alpha n \log\log_\sigma n)$ time the special case where the patterns in the pair have to be of the same length. Lastly, we develop improved versions of the general algorithms for large α by giving an $O((\sigma + \log n)n\sqrt{n}\log_\sigma n)$ time algorithm. The space requirement is only $O(n)$ for these bijective mapping based algorithms.

Furthermore, since primers need to flank the region to be amplified, we also study a natural extension to the problem where the patterns in the pair need to satisfy a set of desired properties and occur at certain positions at a distance at most α, but do not occur α-close anywhere else, in the input string. We modify the bijective mapping based algorithms for this extended problem. Since the restriction can make "short" pattern pairs impossible, we also discuss a variant that allows for arbitrary pattern lengths. We note that for the case of primers, which typically have lengths in the range [17,25], the obtained algorithm runs in $O(\alpha n)$ time and $O(n)$ space.

2. Preliminaries

2.1. Definitions

A *string* $T = t_1 t_2 \cdots t_n$ is a sequence of *characters* from an ordered *alphabet* Σ of size σ. The *length* of string $T = t_1 t_2 \cdots t_n$ is n and is denoted by $|T|$. The *empty* string, denoted by ε, is a string of length 0, that is, $|\varepsilon| = 0$. A *substring* of T is any string $T_{i\ldots j} = t_i t_{i+1} \cdots t_j$, where $1 \leqslant i \leqslant j \leqslant n$. A substring of length k is called a *k-mer*. A *suffix* of T is any substring $T_{i\ldots n}$, where $1 \leqslant i \leqslant n$. A *prefix* of T is any substring $T_{1\ldots j}$, where $1 \leqslant j \leqslant n$. Suffixes and prefixes can be identified by their starting and ending positions, respectively. A *pattern* is a short string over the alphabet Σ. We say that pattern $P = p_1 p_2 \cdots p_k$ *occurs* at position j of *text* string T if and only if $p_1 = t_j, p_2 = t_{j+1}, \ldots, p_k = t_{j+k-1}$. Such positions j are called the occurrence positions of P in T.

A *missing pattern* P (with respect to text T) is such that P is not a substring of T, i.e., P does not occur at any position j of T. Now, let $\alpha > 0$ be a threshold parameter. A *missing pattern pair* (A, B) with threshold α is such that if A (resp. B) occurs at position j of text T, then B (resp. A) does not occur at any position j' of T, such that $j - \alpha \leqslant j' \leqslant j + \alpha$. If (A, B) is a missing pair, we say that A and B do not occur *α-close* in T. These notions are illustrated in Fig. 2.

To make the considered problems non-trivial we assume that the alphabet Σ, and therefore the patterns, consist only of characters appearing in the input text T. Furthermore, we assume that $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. Note that it takes an additive factor of $O(n \log \sigma)$ time to map any ordered alphabet to such Σ.

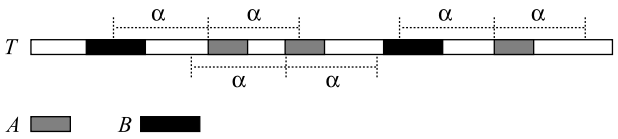

Fig. 2. Missing pattern pair (A, B).

This paper studies the following problems:

Problem 1 *(Single missing pattern discovery problem).* Given a text T, find a shortest pattern A that does not occur in T.

Problem 2 *(Missing pattern pair discovery problem).* Given a text T and a threshold α, find a missing pattern pair (A, B) of minimum total length $|A| + |B|$.

In the preliminary versions of this work [21] and [3], we considered Problem 2 above. However, in the sequel we will show that a single pattern solution to Problem 1 is enough to solve Problem 2 for missing pattern pairs (Corollary 1). Hence we introduce constraints to the lengths of the patterns which make the problem more interesting, that is:

Problem 3 *(Missing pattern pair discovery problem with constraints).* Given a text T and thresholds α, ℓ_1, and ℓ_2, find a missing pattern pair (A, B) of minimum total length such that $|A| \geqslant \ell_1$ and $|B| \geqslant \ell_2$.

Problem 4 *(Same-length missing pattern pair discovery problem).* Given a text T and a threshold α, find a missing pattern pair (A, B) of minimum total length such that $|A| = |B|$.

We also study the corresponding *Missing Pattern* (*Pair*) *Listing* problems, where instead of discovering a *single* solution, one needs to list *all* optimal solutions.

2.2. Data structures

We use some well-known string data structures in our algorithms, such as *keyword tries* and *suffix trees*. Let us briefly recall these structures by following the definitions of [15].

Definition 1 *(Keyword tries).* The *keyword trie* for set $\mathcal{P}$ of patterns is a rooted directed tree $\mathcal{K}$ satisfying the following three conditions:

(1) each edge is labeled with exactly one character,
(2) any two edges out of the same node have distinct labels, and
(3) every pattern P of $\mathcal{P}$ maps to some node v of $\mathcal{K}$ such that the characters on the path from the root of $\mathcal{K}$ to v spell out P, and every leaf of $\mathcal{K}$ is mapped to by some pattern in $\mathcal{P}$.

Definition 2 *(Suffix tries).* The *suffix trie* of text T is a keyword trie for set $\mathcal{S}$, where $\mathcal{S}$ is the set of all suffixes of T.

Definition 3 *(Suffix trees).* The *suffix tree* of text T is the *path-compressed* suffix trie of T, i.e., a tree that is obtained by representing each maximal non-branching path of the suffix trie as a single edge labeled by the concatenation of the labels in the corresponding edges of the suffix trie. The labels of the edges of suffix tree correspond to substrings of T; each edge can be represented as a pair (l, r), such that $T_{l\ldots r}$ gives the label.

Definition 4 *(Sparse suffix trees).* The *sparse suffix tree* of text T is a suffix tree built on a subset $\mathcal{S}'$ of suffixes of T, i.e., a path-compressed keyword trie for set $\mathcal{S}'$.

Fig. 3 shows the suffix tree and a sparse suffix tree of a text $T = ababc$.
Due to [13] we have:

Theorem 1 *(Suffix tree construction). The suffix tree of a text T over alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$ can be constructed in $O(n)$ time and space, where $n = |T|$ and $\sigma = O(n)$.*

A sparse suffix tree can be obtained by pruning the corresponding (full) suffix tree [22,2], that is:

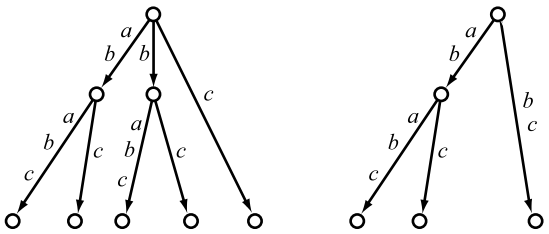

Fig. 3. To the left is the suffix tree of text $T = ababc$ that represents the set $\mathcal{S} = \{ababc, babc, abc, bc, c, \varepsilon\}$ of all the suffixes of T. To the right is the sparse suffix tree for the subset $\mathcal{S}' = \{ababc, abc, bc, \varepsilon\}$ of $\mathcal{S}$.

Theorem 2 *(Sparse suffix tree construction). The sparse suffix tree of a text T with respect to a subset $\mathcal{S}'$ of suffixes of T can be constructed in $O(n)$ time.*

The nodes of all the above-defined trees can be partitioned into two classes: (1) A node is *complete* if it has an edge $e(c)$ for each $c \in \Sigma$ such that the label of edge $e(c)$ starts with character c; (2) otherwise the node is *incomplete*. Let us denote by $label(v, s)$ the concatenation of labels between two nodes v and s. With the *depth* of a node v we mean $|label(root, v)|$.

We sometimes refer to *implicit nodes* of the suffix tree, meaning, in addition to all (explicit) nodes of the suffix tree, also the positions on the edge labels of the suffix tree, as they all correspond to nodes of the corresponding suffix trie.

3. Suffix tree based approach

This section is devoted to showing our suffix tree based algorithm for finding missing patterns. In what follows we describe how a shortest single missing pattern can be found by using suffix trees. Firstly, we show how to solve Problem 1 using suffix tries. Build the suffix trie of T. Among all incomplete nodes of the trie, select the one that has the minimum depth. Let that node be v and let a character that makes the node incomplete be c. Then $label(root, v)c$ is a shortest missing pattern for T. The size of the suffix trie can be $O(n^2)$. However, the same algorithm can be simulated using the suffix tree of T which reduces the running time and working space to $O(n)$. Instead of scanning through all the implicit nodes of the suffix tree, we can check the explicit nodes for incompleteness and for each edge whose label is longer than 1, we know that the implicit node corresponding to the first character on the label is incomplete. Since by Theorem 1, the suffix tree of T can be built in $O(n)$ time and space, we have the following result.

Theorem 3. *For any text string of length n, the proposed algorithm solves Problem 1 in $O(n)$ time and space.*

It is also immediate that the algorithm can be extended to list the set S of all the solutions in the optimal $O(n + |S|)$ time.

3.1. Basic properties

The topic of the paper is missing pattern *pair* discovery. However, some aspects of the single-pattern solution can be exploited, e.g., the following observation is useful.

Observation 1 *(Monotony property). Let v be a node of the suffix tree of text T, and let e be an edge out of v which is labeled $L = l_1 l_2 \cdots l_p$. Then, string $label(root, v)l_1$ occurs in T exactly at the same positions as string $label(root, v)L_{1\ldots i}$ for any $1 < i \leqslant p$.*

The problems of interest are Problems 3 and 4. To see why Problem 2 is not interesting as such, and to motivate the more refined problem statements, the following observations state that the single-pattern solution is enough when no constraints are set to the pattern lengths.

Lemma 1 *(Substring Property 1). Let (A, B), $|A| \geqslant |B|$, be a solution to Problem 2. It holds that either,*

(1) *Both A and B are substrings of the input text*; *or*
(2) *Pattern A is a single missing pattern and B is an empty string.*

Proof. Let (A, B) be a solution to the missing pairs problem such that A is not a substring of the text. Then (A, ε) is also a missing pair. Since $|A| + |\varepsilon| = |A| \leqslant |A| + |B|$, pair (A, B) cannot be the shortest missing pair, unless B is the empty string, in which case A is a single pattern solution. The case where B is not a substring is symmetric. □

Corollary 1. *Problem 2 on a text of length n can be solved in $O(n)$ time, when $\alpha \geqslant |A| - 1$, by finding a pair (A, ε) where A is a shortest missing pattern.*

Proof. Let A be an optimal solution to Problem 1. Then (A, ε) is a missing pair of total length $|A|$. If (A, ε) is not an optimal solution to Problem 2, then there must be a missing pair (A', B) such that $|A'| + |B| < |A|$. By Lemma 1 both A' and B are substrings of the text, otherwise A could not be the solution to Problem 1, contradicting the assumption. Let i be an arbitrary occurrence position of A' in the text. By the definition of missing pair, B cannot occur at position $i + |A'|$, unless $\alpha < |A'|$. That is, the concatenation $A'B$ is a missing pattern of length $|A'| + |B| < |A|$. This contradicts the assumption of A being the minimum length missing pattern, and hence (A, ε) is a solution to Problem 2 when $\alpha \geqslant |A'|$ (and thus when $\alpha \geqslant |A| - 1$ while $|A| > |A'|$). Missing pattern A can be computed in time $O(n)$ by Theorem 3. □

We will get back to special case $\alpha < |A| - 1$ later. Note that the listing version of Problem 2 cannot be solved as easily; we only know that the shortest missing pattern pair must be of length $|A|$.

Let us now concentrate on solving Problems 3 and 4, and the listing versions of all the missing pattern pair problems. We will use the following lemmas.

Lemma 2 *(Substring Property 2). Let (A, B) be a solution to Problem 3 (or to Problem 4). It holds that either,*

(1) *Both A and B are substrings of the text*; *or*
(2) *There is a missing pair (A', B') that can be computed in $O(|A| + |B|)$ time from a shortest single missing pattern such that $|A'| = |A|$ and $|B'| = |B|$.*

Proof. Let (A, B) be a solution to Problem 3 such that A is not a substring of the text. One can replace A with a pattern A' computed from a shortest single missing pattern padded to length $|A'|$ with arbitrary alphabet symbols, when necessary, and B with any pattern B' of length $|B|$. Since A' is also missing and $|A'| + |B'| = |A| + |B|$, pair (A', B') is a solution to Problem 3. The case where B is not a substring is symmetric. The same arguments prove the lemma for Problem 4. □

Lemma 3. *If $\sigma^k \geqslant n$ for some $k > 1$, then there must be a single missing pattern of length $k > 1$.*

Proof. There are at most $n - k + 1 < n$ different k-mers in a string T of length n. Since there are $\sigma^k \geqslant n$ distinct strings of length k, there must be some k-mer $X \in \Sigma^k$ that is not a substring of T. □

3.2. Basic algorithm

We now present an algorithm for Problem 2. Although this problem was just shown to be easily solvable via the single pattern solution (Lemma 1), the algorithm derived below is more general and can be adjusted with small modification to all the other problem variants. These modifications will be discussed in the end.

Let V be the set of all nodes of the suffix tree of text T, and let $\mathcal{P}$ be the set of strings obtained by adding to each $label(root, v)$, $v \in V$, all starting characters of labels on the out edges of v. It is easy to see that $|\mathcal{P}| \leqslant 2n - 1$. That is, the size of $\mathcal{P}$ is at most the number of nodes in the tree. Finally, let $Occ(P)$ be the list of occurrences of pattern $P \in \mathcal{P}$ in T; it can be obtained in time $O(|Occ(P)|)$ from the suffix tree.

Recall that we are interested in finding a missing pair (A, B). Let us choose as A a string from $\mathcal{P}$. Our goal is to choose B so that (A, B) will be a missing pair. As A is now fixed, we try to choose B of minimum length. Let us, for now, assume that we have found pattern B of minimum length such that (A, B) is a missing pair. The crucial observation is that if we repeat this process for all $A \in \mathcal{P}$, we can choose among all the missing pairs found so far, the one where the sum $|A| + |B|$ is minimized. The correctness of this procedure follows directly from Observation 1 and Lemma 1.

What is left is to explain how to choose B of minimum length so that (A, B) will be a missing pair. This is done as follows. Let us define a set $Zone(A, \alpha)$:

$$Zone(A, \alpha) = \bigcup_{j \in Occ(A)} [j - \alpha, j + \alpha].$$

We have the following observation:

Observation 2. *B is a prefix of any suffix $T_{j'\ldots n}$ such that $j' \in Zone(A, \alpha)$ if and only if pair (A, B) occurs α-close in T.*

Now, building the sparse suffix tree over suffixes $T_{j'\ldots n}$, $j' \in Zone(A, \alpha)$, we can choose B exactly as mentioned in the proof of Theorem 3: Among all incomplete implicit nodes of the sparse suffix tree, select the one that has the minimum depth. Let that node be u and let a character that makes the node incomplete be d. Then $B = label(root, u)d$. The algorithm is illustrated in Fig. 4.

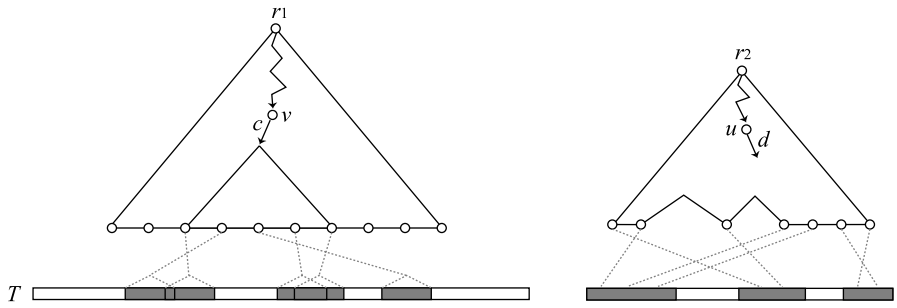

Fig. 4. Illustration of the algorithm to find a missing pair (A, B), where $A = label(r_1, v)c$, $B = label(r_2, u)d$, r_1 is the root of the full suffix tree, and r_2 is the root of the sparse suffix tree corresponding to A.

Now we have the following theorem.

Theorem 4. *For any text string of length n, the proposed algorithm solves Problems 2, 3, and 4 in $O(n^2)$ time and $O(n)$ space.*

Proof. The correctness of the algorithm for solving Problem 2 should be clear from the above discussion. The time complexity follows from the facts that the size of $\mathcal{P}$ is at most $2n - 1$, and for each $A \in \mathcal{P}$ we use $O(n)$ time for constructing the set $Zone(A, \alpha)$ and the corresponding sparse suffix tree; To construct $Zone(A, \alpha)$ in linear time, one should first mark in a bit-vector of length n all suffixes in $Occ(A)$. Then for each marked suffix j, one should mark in some other bit-vectors the starting point $j - \alpha$ and the end point $j + \alpha$ of the influence region. Finally, scanning from left to right one can maintain a counter to know at each text position j' whether it is inside some influence region or not, i.e., whether it should be included in the sparse suffix tree or not. As mentioned earlier, the sparse suffix tree can be obtained from the full suffix tree in $O(n)$ time. Overall, we have $O(n \times n) = O(n^2)$ time. At each phase of the algorithm, we use $O(n)$ space. Extending the algorithm to solving Problems 3 and 4 is easy without increasing the computational complexities; For the latter, accept only patterns B of length $|A|$. For the former, consider only patterns $A \in \mathcal{P}$ having length at least ℓ_1, and accept patterns B of length at least ℓ_2. □

It is also easy to modify the algorithm to list all optimal solutions for any of the Problems 2, 3, and 4.

3.3. Improved algorithm

In this section, we show an improved algorithm in the case where α is small. First, we observe that we can select pattern A near the root of the suffix tree because of Lemma 3; we can restrict to the cases $|A|, |B| \leqslant \log_\sigma n$, as the solution can otherwise be derived from a single pattern solution.

Let $\mathcal{P}^{\leqslant q}$ be a subset of $\mathcal{P}$ such that all strings in $\mathcal{P}^{\leqslant q}$ are at most of length q. Now, we make the following observation:

Observation 3. *For each suffix $T_{j\ldots n}$, there are at most $q = \log_\sigma n$ strings $A \in \mathcal{P}^{\leqslant q}$ such that $j \in Occ(A)$.*

A direct consequence of Observation 3 is that the overall size of sparse suffix trees corresponding to strings $A \in \mathcal{P}^{\leqslant q}$ is at most $O(\alpha n \log_\sigma n)$; each suffix can belong to at most $(2\alpha + 1)\log_\sigma n$ different sparse suffix trees, and the size of a sparse suffix tree is proportional to the number of suffixes it contains.

Now, we can build the sparse suffix trees incrementally in linear time in their overall size as follows: make a depth-first search (DFS) on the full suffix tree limited to depth $\log_\sigma n$. Let SST_v be the sparse suffix tree corresponding to an internal node v; more formally, SST_v is the sparse suffix tree of the suffixes at positions $j \in Zone(A, \alpha)$, where $A = label(root, u)c$, u is the parent of v, and c is the first character of the edge label from u to v. Let g be the child node of v to which we are proceeding in the DFS search. We make the observation that the sparse suffix tree SST_g corresponding to node g will contain a subset of suffixes represented by SST_v; we can prune SST_v to construct SST_g. To manage the incremental computation efficiently, we show in the next lemma that SST_g can be constructed from SST_v in linear time in the size of SST_v. To make this possible, we need to attach some additional information to the sparse suffix trees: We use *threaded* sparse suffix trees, where the leaves (suffixes) of the tree are linked together in a double linked list in increasing order of the suffix positions, and each leaf has a pointer to the corresponding leaf of the full suffix tree.

Lemma 4. *Let SST_v be the threaded sparse suffix tree corresponding to a node v of the full suffix tree (in the sense defined above). Then, the threaded sparse suffix tree SST_g corresponding to the child g of v can be constructed in linear time in the size of SST_v.*

Proof. The algorithm is as follows. We make a copy of SST_v and prune it (i.e., delete extra leaves) to construct SST_g. Let us simply use SST_v to denote the copy of it. The construction has three phases; (i) we mark all leaves (suffixes) of SST_v that

are contained in the subtree of g in the full suffix tree, (ii) we mark all leaves of SST_v whose suffix positions are within α distance from the ones marked at phase (i), and (iii) we delete all unmarked leaves of SST_v to construct SST_g.

Phase (iii) is trivial; as a leaf is deleted (making some constant time local updates to the tree) we redirect the links between suffix positions to retain the threaded structure. In phase (ii) we extend the effect of the suffixes marked in phase (i) by scanning through the double linked list once from first to last and once from last to first. For phase (i) recall that the leaves of SST_v have pointers to the corresponding leaves of the full suffix tree. We reverse these pointers, so that we have pointers from some leaves of the full suffix tree to SST_v. Then we go through the leaves in the subtree of g, and follow the pointers from these leaves marking the corresponding leaves of SST_v. This concludes phase (i).

It is clear that after steps (i), (ii), and (iii), the remaining tree corresponds to SST_g, and the construction time is linear in the size of the tree SST_v. □

After noticing that the threaded version of the full suffix tree is easy to obtain in linear time in its size, we get by induction using Lemma 4 the following result.

Theorem 5. *For any text string of length n, the proposed algorithm solves Problems 2, 3, and 4 in $O(\min\{n^2, \alpha n \log n\})$ time and $O(n \log n)$ space on a constant alphabet.*

Proof. Lemma 4 states that we use linear time in the size of the parent sparse suffix tree to construct the child sparse suffix tree. Each node of the full suffix tree can have at most σ children, and hence we can use time at most σ times the size of each sparse suffix tree. The claimed bound follows by taking the minimum of the trivial $O(n^2)$ bound and of the α-dependent bound $O(\alpha n \log n)$ on the overall size of sparse suffix trees, assuming σ is constant.

The maximum space usage during the algorithm follows from the fact that we need to store at most $\log_\sigma n$ different sparse suffix trees at the same time during the DFS to manage the incremental computation. Extending the algorithm to solving Problems 3 and 4 is identical to the base algorithm. □

Remark. The constant multiplicative factor σ occurring in the proof of the above theorem can be reduced to 1 by organizing the edges of each node of the full suffix tree in a balanced tree; we can build temporary sparse suffix trees for the nodes of each balanced tree. The overall size of the trees is $O(\log \sigma \alpha n \log_\sigma n) = O(\alpha n \log n)$, and each tree is now scanned through only a constant number of times; the time requirement is thus reduced to $O(\alpha n \log n)$ without any dependency on σ. The space requirement is increased from $O(n \log_\sigma n)$ to $O(n \log_2 n)$.

The improved algorithm solves the listing versions of the problems as well. Now we have also covered the special case $\alpha < |A| - 1$ for Problem 2, as for case $\alpha < |A| - 1 < \log_\sigma n$ the time requirement $O(\alpha n \log n)$ turns into the following.

Corollary 2. *Problem 2 on a text of length n can be solved in $O(n \log^2 n)$ time, when $\alpha < |A| - 1$, where A is a shortest single missing pattern.*

4. Bijective mapping based approach

In this section, we present simpler algorithms for finding a shortest single missing pattern and missing pattern pair. The algorithms are based on a natural bijective mapping of patterns to integers. For missing pattern pairs we present two algorithms, one with running time that depends on α and the other suitable for large α. These algorithms will use, as main routines, procedures which find a missing pattern pair where each pattern is of pre-defined length.

4.1. Finding single missing patterns

Recall the algorithm of Theorem 3 that finds a single shortest missing pattern. It uses the suffix tree data structure to compactly enumerate all patterns found in the input string. Here, we use Lemma 3 to limit the number of considered patterns. The algorithm works by computing a boolean table of all patterns of length $\lfloor \log_\sigma n \rfloor$ that occur in the input text T using a natural bijective mapping (hashing) of the patterns to the integers $0, 1, \ldots, \sigma^{\lfloor \log_\sigma n \rfloor} - 1$. This can be done in linear time by scanning the input string from left to right using the established technique of computing the hash of pattern Yb knowing the hash of pattern aY (here, $Y \in \Sigma^k$, for some $k \geqslant 0$, and $a, b \in \Sigma$). Let the hash $h(X)$ for a pattern $X = x_1 \cdots x_{|X|}$ be:

$$h(X) = \sum_{i=1}^{|X|} x_i \sigma^{|X|-i}.$$

Then, the hash of Yb can be calculated from that of aY in constant time since (see for example [23]),

$$h(Yb) = \sigma\left(h(aY) - a\sigma^{|Y|}\right) + b.$$

Let E_k be the boolean table for patterns of length k such that $E_k[\hat{h}]$ is true if and only if the pattern of length k with hash $\hat{h}$ occurs in T. By analyzing consecutive runs of missing pattern pairs in E_k we efficiently find the shortest missing pattern pair as follows.

Note that the size of E_k, denoted by $|E_k|$, is σ^k. Furthermore, we have $E_k[h] = \bigvee_{i=0}^{\sigma-1} E_{k+1}[\hat{h}\sigma + i]$. That is, a pattern of length k is missing from T if all possible suffix extensions by symbol in Σ are also missing. Therefore, we can compute E_k from E_{k+1} in $O(\sigma|E_{k+1}|)$ time. By computing $E_{\lfloor \log_\sigma n \rfloor}$ in $O(n)$ time, we can compute all E_k, for $k < \lfloor \log_\sigma n \rfloor$, in $O(\sum_{k=2}^{\lfloor \log_\sigma n \rfloor - 1} \sigma^{k+1}) = O(n)$ time and space, and therefore find a shortest missing pattern of length $\leqslant k$, if one exists.

If all patterns of length $\lfloor \log_\sigma n \rfloor$ occur in T, then the shortest missing pattern is of length $\lceil \log_\sigma n \rceil$. In this case we can find a representative by computing the first n entries of the boolean table $E_{\lceil \log_\sigma n \rceil}$. We obtain the next theorem.

Theorem 6. *For any text string of length n, the proposed algorithm solves Problem 1 in $O(n)$ time and space (bits).*

4.2. α-dependent algorithms

4.2.1. Finding missing pairs of fixed lengths

We now present an $O(\alpha n)$ time and $O(n)$ space algorithm that finds a missing pattern pair (A, B), where the lengths of A and B are given as input parameters and are at most $\lfloor \log_\sigma n \rfloor$. The algorithm serves as a basis for the subsequent algorithms for finding the missing pattern pairs. Here, we only focus on finding missing pattern pairs (A, B) such that A and B have length less than the length of a shortest missing single pattern (see Lemmas 1 and 2). In what follows, we let $\ell^* \leqslant \lceil \log_\sigma n \rceil$ be the length of a shortest missing pattern.

Let $|A| = a$ and $|B| = b$ and assume, without loss of generality, $a \geqslant b$. Let $N_1 = \sigma^a$ and $N_2 = \sigma^b$ be the number of distinct patterns of length a and b, respectively. (Clearly, $n > N_1 \geqslant N_2$.) The following algorithm heavily uses the bijective mapping, $h(\cdot)$, of patterns to integers outlined in Section 4.1.

Bijective Mapping Algorithm 1.

1. Let L be an array of length N_1, where $L[h(A)]$ is the list of occurrences in T of the pattern A of length a. That is, $L[h(A)] = Occ(A)$.
2. Compute an array H of length $n - b + 1$ such that $H[j] = h(B)$, where B is the pattern of length b that occurs at position j of T.
3. For each $A \in \Sigma^a$ count the number of distinct patterns B of length b that are α-close to A. We do this by maintaining a boolean table of the distinct patterns B that are α-close to A.

At each iteration we perform the following sub-steps. Let $\hat{h} = h(A)$.

(i) For each occurrence in $L[\hat{h}]$ of pattern A, we mark in a table M of size N_2 all patterns of length b that occur at distance at most α by scanning the corresponding positions of the array H.
(ii) When a pattern of length b is seen for the first time we increase a counter. The counter is set to 0 at the beginning of each iteration.
(iii) The iteration ends when the maintained counter becomes equal to N_2 to indicate that all patterns of length b are α-close to A, or when all of $L[\hat{h}]$ is processed. At the end of an iteration, if the counter is less than N_2, we scan M to find a missing pattern pair and the algorithm terminates.

Theorem 7. *Given integer parameters $a \leqslant \lfloor \log_\sigma n \rfloor$ and $b \leqslant \lfloor \log_\sigma n \rfloor$, Bijective Mapping Algorithm 1 finds a missing pattern pair (A, B) such that $|A| = a$ and $|B| = b$, if one exists, in $O(\alpha n)$ time and $O(n)$ space.*

Proof. The correctness of the algorithm follows from the fact that for each pattern of length a, we exhaustively enumerate all patterns of length b that occur α-close. We now analyze its performance. Step 1 of the algorithm can be performed in $O(n)$ time by scanning T from left to right. Compute the hash $h(\cdot)$ of the pattern at position i from that of position $i-1$ and append i to the list $L[h(\cdot)]$. The total size of all lists is $O(n - a + 1)$. The array H in Step 2 can be computed in a similar fashion and takes $O(n - b + 1)$ space. An iteration of Step 3 takes $O(\alpha|L[\hat{h}]|)$ time for a total of $O(\alpha n)$ time and an additional $O(N_2) = O(n)$ space. We conclude the algorithm will output a missing pair (A, B) with the desired pattern lengths, if such pair exists, in $O(\alpha n)$ time and $O(n)$ space. □

4.2.2. Finding missing pattern pairs of the same length

We combine the algorithm from the previous subsection and the following observation to obtain an efficient algorithm to solve Problem 4 where we are required to find missing pairs consisting of patterns of the same length.

Observation 4. *If a pattern pair (A, B) is missing, the pair (C, D), where A is a substring of C and B is a substring of D, is also missing.*

We are now ready to state the following corollary of Theorem 7.

Corollary 3. *Problem 4 can be solved in $O(\alpha n \log\log_\sigma n)$ time and $O(n)$ space.*

Proof. Recall that there exists a missing pattern pair (A, B), where $a = b = \ell^* \leqslant \lceil \log_\sigma n \rceil$. Therefore, such a missing pair can be found in linear time and space by Theorem 6. In order to find a pair of minimum total length, we can do binary search on the pattern length $1, \ldots, (\ell^* - 1)$ and apply Bijective Mapping Algorithm 1 for each length. From the monotonicity property of Observation 4, we are guaranteed to output a shortest missing pattern pair of the same length in $O(\alpha n \log\log_\sigma n)$ time and $O(n)$ space. □

4.2.3. Finding missing pattern pairs of different length

We now consider Problem 3 where the two patterns in the missing pair are not necessarily of the same length. We obtain the following corollary of Theorem 7.

Corollary 4. *Problem 3 can be solved in $O(\alpha n \log_\sigma n)$ time and $O(n)$ space.*

Proof. By applying Bijective Mapping Algorithm 1 for all choices of $a \in \{\ell_1, \ell_1 + 1, \ldots, \ell^* - 1\}$ and $b \in \{\ell_2, \ell_2 + 1, \ldots, \ell^* - 1\}$, we obtain an algorithm which runs in $O(\alpha n \log^2_\sigma n)$ time and $O(n)$ space. We improve the running time to $O(\alpha n \log_\sigma n \log\log_\sigma n)$ by enumerating all choices of a and performing binary search on b. The correctness of the algorithm follows from Observation 4. We further improve the running time using the following claim which follows directly from the same observation.

Claim 1. *Let (A, B) and (C, D) be shortest missing pattern pairs such that $|A| = a$ and $|C| = c$ for fixed a and c where $a \leqslant c$. Then, $|B| \geqslant |D|$.*

Therefore, by enumerating all choices of a in increasing order, we can consider only choices of b in non-increasing order. Since there are $O(\log_\sigma n)$ choices for each a and b, we obtain the desired running time. □

4.3. α-independent algorithms

We now present algorithms with running time independent of the threshold parameter α suitable for finding missing pattern pairs with total length at most $\ell^* \leqslant \lceil \log_\sigma n \rceil$. The algorithms follow similar framework to those presented in Section 4.2. We first give a base algorithm that finds for all pairs (A, B), where the lengths of A and B are given as input parameters, the smallest α_{AB} such that A and B occur α_{AB}-close. Now, a pattern pair (A, B) is missing if and only if $\alpha_{AB} > \alpha$. We then extend the base algorithm in order to find all shortest missing pattern pairs where each pattern is a substring of the input text (see Lemma 1 and its corollary). To obtain efficiency, we take advantage of the fact that there are not too many pattern pairs (A, B) of total length ℓ^*. More precisely, for given lengths of A and B, there are at most $\sigma^{\lceil \log_\sigma n \rceil} < \sigma n$ such pairs.

The steps of the base algorithm are as follows.

Bijective Mapping Algorithm 2.

1. (i) Let L be an array of length N_1, where $L[h(A)]$ is the list of occurrences in T of the pattern A of length a. That is, $L[h(A)] = Occ(A)$.
   (ii) Let R be an array of length N_2, where $R[h(B)]$ is the list of occurrences in T of the pattern B of length b. That is, $R[h(B)] = Occ(B)$.
2. For each pattern pair (A, B), merge the lists of occurrence positions $L[h(A)]$ and $R[h(B)]$ (which are sorted by construction) to find the closest occurrence of A and B and therefore α_{AB}.

Theorem 8. *Given integer parameters a and b such that $a + b \leqslant \lceil \log_\sigma n \rceil$, Bijective Mapping Algorithm 2 finds a missing pattern pair (A, B) such that $|A| = a$ and $|B| = b$, if one exists, in $O((\sigma + \log n)n\sqrt{n})$ time and $O(n)$ space.*

Proof. The correctness of the algorithm follows from the fact that for each pattern of length a, we exhaustively enumerate all patterns of length b to find one that does not occur α-close. We now analyze its performance. The algorithm clearly requires $O(n)$ space, and we claim it takes $O((\sigma + \log n)n\sqrt{n})$ time. Step 1 of the algorithm can be performed in $O(n)$ time by scanning T from left to right. Consider Step 2. For a given pattern, we will call its list of occurrence positions *long* if it has length at least $\sqrt{n}$. We note that there are at most $\sqrt{n}$ long lists in L since the total length of all lists is at most n. Similarly, there are at most $\sqrt{n}$ long lists in R. All pairs of lists that are not long can be merged in $O(\sigma n\sqrt{n})$ time using merge sort since there are $O(\sigma n)$ such pairs. Let I be the set of indices of long lists in L, i.e., for all $\hat{h} \in I$, $|L[\hat{h}]| \geqslant \sqrt{n}$. Fix $\hat{h} \in I$. The

list $L[\hat{h}]$ can be merged using binary search with all lists in R in time proportional to $\sum_{\hat{h}'} |R[\hat{h}']| \log |L[\hat{h}]| = O(n \log |L[\hat{h}]|)$. Summing over $\hat{h} \in I$ we obtain $n \sum_{\hat{h} \in I} \log |L[\hat{h}]| = O(n\sqrt{n} \log n)$ because $|I| \leqslant \sqrt{n}$ and each list is of length at most n. Applying symmetric argument for the long lists in R we obtain the desired running time. □

The next result follows from Lemma 1 and its corollary. In order to find a shortest pattern pair (A, B) with total length ℓ^* where A and B are substrings of the input string, we need to consider $O(\ell^*)$ combinations of lengths for A and B.

Corollary 5. *Problem 2 can be solved in $O((\sigma + \log n)n\sqrt{n} \log_\sigma n)$ time and $O(n)$ space.*

Note that the algorithms of this section can be extended to list all the optimal solutions in addition to only finding a single missing pair.

5. Extensions to the missing pattern pair problem

In this section we discuss the following two extensions to the problem of finding missing pattern pairs of fixed lengths. First, we show how to find missing pairs when the patterns are restricted to occur at certain regions of the input string T. The restriction can be determined based on the region we would like to amplify or/and biologically motivated constrains such as CG content or free energy constraints, e.g., see [29]. The latter type of constraints can be computed for patterns of given length in $O(n)$ for all positions in T (under some simplifications, e.g., see [10]).

Next, in addition, we allow the patterns to be of length greater than $\lfloor \log_{\sigma n} \rfloor$. This is necessary because the patterns need to occur in the input string. We describe the required changes to the bijective mapping algorithm of Section 4.2.1, and then state how the extended problems generalize to Problems 3 and 4.

5.1. Localized patterns of given length

Given lengths a and b, let P_a and P_b be two subsets of positions in the input string T. We are interested in finding a pattern pair (A, B) such that $|A| = a$, $|B| = b$, and there exist $j \in Occ(A) \cap P_a$ and $j' \in Occ(B) \cap P_b$ such that $|j - j'| \leqslant \alpha$. Furthermore, if patterns A and B occur α-close at positions j and j' then $j \in P_a$ and $j' \in P_b$.

The sets P_a and P_b can be specified as interval lists or bit-tables. For simplicity we assume the latter representation, which can be obtained from the interval lists in $O(n)$ time and space (the conversion can be done using the same technique as in the proof of Theorem 4).

We modify the algorithm of Section 4.2.1 as follows. We restrict the occurrence lists of A in L only to those in P_a in a straightforward manner. In the same fashion, in Step 3, we count for each pattern A, the distinct patterns B that occur at distance at most α and do not start in positions in P_b. If there is a pattern missing, we do an additional pass to look for an α-close unmarked pattern that starts in P_b.

It is not hard to see that with the described modifications the space requirement of the algorithm remains $O(n)$. To show that the running is $O(\alpha n)$ we need to be able to initialize table M (Step 3) in $O(\alpha n)$ time for all iterations. Note that before, after an iteration for pattern A, either all entries in M are marked or we declare a pattern pair to be missing. Here, to initialize M efficiently, we need to keep track of the entries of M that are marked ($O(n)$ entries per iteration; $O(\alpha n)$ entries in total) in the previous iteration and initialize only those entries. Alternatively, we can repeat the iteration but initializing the corresponding entries. We conclude the running time remains $O(\alpha n)$ after the modifications.

5.2. Long patterns

Since patterns are restricted to occur in the input string T, there are at most n candidate patterns for each A and B irrespective to their given length. For patterns of length greater than $\lfloor \log_\sigma n \rfloor$, we can maintain the same framework of the algorithm of Section 4.2.1 given a suitable (hash) function mapping valid A and B patterns to integers 0 to $O(n)$ corresponding to lists L (Step 1) and H (Step 2). We obtain such a mapping by computing the suffix tree of T and using the node indices corresponding to the patterns of length $|A|$ and $|B|$ in a standard way (see for details [15]). Computing the suffix tree only requires additional $O(n)$ time and space (Theorem 1).

5.3. Generalized pattern pair problem

We are now ready to state the following theorem.

Theorem 9. *For any string of length n, the proposed algorithms solve the generalized missing pattern pair discovery problem in*

- *$O(\alpha n \kappa)$ time when the patterns are of the same length*;
- *$O(\alpha n \kappa^2)$ time when the patterns are allowed to have different lengths,*

where κ is the total length of the output pair. In both cases the space requirement is $O(n)$.

Table 1
Unordered missing pattern pairs in both the human and baker's yeast genomes for $k = 8$. The reverse complements of the shown pattern pairs are also missing.

Missing pairs	Yeast α_{AB}	Human α_{AB}
(AATCGACG, CGATCGGT)	5008	6458
(CCGATCGG, CCGTACGG)	5658	6839
(CGACCGTA, TACGGTCG)	13 933	7585
(CGACCGTA, TCGCGTAC)	5494	5345
(CGAGTACG, GTCGATCG)	5903	8090
(CGATCGGA, GCGCGATA)	6432	6619

Table 2
Single missing patterns of length 11 from the human genome. The reverse complements of the shown patters are also missing.

Missing patterns		
ATTTCGTCGCG	CGGCCGTACGA	CGCGAACGTTA
CCGAATACGCG	CGTCGCTCGAA	CGTTACGACGA
CCGACGATCGA	CGACGCGATAG	GCGTCGAACGA
CGCGTCGATAG	CGATTCGGCGA	TATCGCGTCGA

Proof. Note that because of the condition that patterns must occur α-close at specific positions (and based on their length and properties), Observation 4 might not hold. We therefore need to run the extended version of the algorithm of Section 4.2.1 for all possible combinations of pattern lengths up to κ. □

6. Experiments

We have performed tests with the baker's yeast (*Saccharomyces cerevisiae*) genome and the human genome.[2]

We set the distance α to 5000, which is a realistic value in terms of the speed of the polymerase reaction and duration of the PCR cycle. We then searched for shortest missing pattern pairs of the same length k. There were solutions with $k = 8$ for the yeast genome (i.e. both patterns of the pair are of length 8), in fact there were over 16 million such pairs. From the human genome data, we found 238 missing pattern pairs with $k = 8$. This is an interesting result, since the human genome is about 250 times larger than the yeast genome. Of the 238 pattern pairs, 20 pairs are missing from both the human and the baker's yeast genome. Table 1 summarizes these missing pairs and the shortest distance between the patterns (or their reverse complements) of each pair in the corresponding genomes. For reference, the shortest single missing patterns from the human genome are of length 11 and are listed in Table 2. This is also surprising since the human genome length is roughly equal to 4^{16}.

The program needed about 3 hours to process the baker's yeast genome on a 1 GHz machine, and about 30 hours for the human genome. The stop condition of Step 3 of the algorithm of Section 4.2.1, namely when all pattern pairs are discovered for the current pattern, provides a significant optimization in practice which allows the software to run only 10 times slower (rather than 250 times) for the human genome compared to the yeast genome.

7. Conclusions

This paper presented efficient algorithms to solve the missing pattern discovery problems. Table 3 summarizes the results of this paper.

We implemented Bijective Mapping Algorithm 1 and made experiments for the human genome and the baker's yeast genome, and we succeeded in finding shortest missing pairs of length 8 for both human and yeast genomes. In addition, we studied an extended version of the problem where patterns in the pair occur at certain positions at a distance at most α, but do not occur α-close anywhere else, in the input string.

Independently of our work, Li [25] proposed an algorithm that solves the problem of listing all the shortest missing pattern pairs in $O(\min\{\alpha n \log n, n^{3/2}\})$ time. The algorithm assumes that the alphabet size σ is constant.

As a generalization of the missing pattern discovery problem, the following problem that allows mismatches is worth to consider: Given string T, distance α, and error parameter e, find pattern pair (A, B) such that any occurrence of A and B within e mismatches in T is not α-close. [28] presented some algorithms to discover structured motifs with errors in the Hamming distance metric. Since the algorithms of [28] and our algorithms in Section 3 are both based on suffix trees, it might be possible to solve the above general missing pattern discovery problem by combining these approaches.

[2] Available at ftp://ftp.ensembl.org/pub/current_human/.

Table 3
Summary of results for finding missing pairs of patterns. In the case of the suffix tree algorithms, the results for patterns of different length hold for the case when patterns are of the same length too.

Algorithm	Time	Space
Patterns of same length		
Bijective Mapping Algorithm 1	$O(\alpha n \log\log_\sigma n)$	$O(n)$
Patterns of different length		
Basic Suffix Tree Algorithm	$O(n^2)$	$O(n)$
Improved Suffix Tree Algorithm	$O(\min\{n^2, \alpha n \log n\})$	$O(n \log n)$
Bijective Mapping Algorithm 1	$O(\alpha n \log_\sigma n)$	$O(n)$
Bijective Mapping Algorithm 2	$O((\sigma + \log n)n\sqrt{n}\log_\sigma n)$	$O(n)$

Acknowledgements

The authors would like to thank Mr. K. Kataja and Dr. R. Satokari. Mr. Kataja was the first one to bring the adapter primer selection problem to our attention and Dr. Satokari has advised us on the biotechnological issues.

References

[1] A. Amir, A. Apostolico, M. Lewenstein, Inverse pattern matching, Journal of Algorithms 24 (2) (1997) 325–339.
[2] A. Andersson, N.J. Larsson, K. Swanson, Suffix trees on words, Algorithmica 23 (3) (1999) 246–260.
[3] S. Angelov, S. Inenaga, Composite pattern discovery for pcr application, in: Proceedings of the 12th International Symposium on String Processing and Information Retrieval (SPIRE 2005), in: Lecture Notes in Computer Science, vol. 3772, Springer-Verlag, 2005, pp. 167–178.
[4] A. Apostolico, Pattern discovery and the algorithmics of surprise, in: Proceedings of the NATO Advanced Study Institute on Artificial Intelligence and Heuristic Methods in Bioinformatics, in: NATO Science Series, vol. 183, 2003, pp. 111–127.
[5] H. Arimura, S. Arikawa, S. Shimozono, Efficient discovery of optimal word-association patterns in large text databases, New Generation Computing 18 (2000) 49–60.
[6] H. Arimura, H. Asaka, H. Sakamoto, S. Arikawa, Efficient discovery of proximity patterns with suffix arrays (extended abstract), in: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001), in: Lecture Notes in Computer Science, vol. 2089, Springer-Verlag, 2001, pp. 152–156.
[7] R.A. Baeza-Yates, Searching subsequences (note), Theoretical Computer Science 78 (2) (1991) 363–376.
[8] H. Bannai, H. Hyyrö, A. Shinohara, M. Takeda, K. Nakai, S. Miyano, An $O(N^2)$ algorithm for discovering optimal Boolean pattern pairs, IEEE/ACM Transactions on Computational Biology and Bioinformatics 1 (4) (2004) 159–170.
[9] H. Bannai, S. Inenaga, A. Shinohara, M. Takeda, S. Miyano, Efficiently finding regulatory elements using correlation with gene expression, Journal of Bioinformatics and Computational Biology 2 (2) (2004) 273–288.
[10] K.J. Breslauer, R. Frank, H. Blocker, L. Marky, Predicting DNA duplex stability from the base sequence, Proceedings of the National Academy of Sciences of the United States of America 83 (11) (1986) 3746–3750.
[11] A.M. Carvalho, A.T. Freitas, A.L. Oliveira, M.-F. Sagot, A highly scalable algorithm for the extraction of cis-regulatory regions, in: Proceedings of the 3rd Asia Pacific Bioinformatics Conference (APBC 2005), Imperial College Press, 2005, pp. 273–282.
[12] E. Eskin, P.A. Pevzner, Finding composite regulatory patterns in DNA sequences, Bioinformatics 18 (2002) S354–S363.
[13] M. Farach, Optimal suffix tree construction with large alphabets, in: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS 1997), IEEE Computer Society, 1997, pp. 137–143.
[14] L. Gąsieniec, P. Indyk, P. Krysta, External inverse pattern matching, in: Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM 1997), 1997, pp. 90–101.
[15] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.
[16] G. Hampikian, T. Andersen, Absent sequences: nullomers and primes, in: Proceedings of the Pacific Symposium on Biocomputing 2007, 2007, pp. 355–366.
[17] M. Hirao, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, A practical algorithm to find the best subsequence patterns, in: Proceedings of the 3rd International Conference on Discovery Science (DS 2000), in: Lecture Notes in Artificial Intelligence, vol. 1967, Springer-Verlag, 2000, pp. 141–154.
[18] M. Hirao, S. Inenaga, A. Shinohara, M. Takeda, S. Arikawa, A practical algorithm to find the best episode patterns, in: Proceedings of the 4th International Conference on Discovery Science (DS 2001), in: Lecture Notes in Artificial Intelligence, vol. 2226, Springer-Verlag, 2001, pp. 435–440.
[19] S. Inenaga, H. Bannai, H. Hyyrö, A. Shinohara, M. Takeda, K. Nakai, S. Miyano, Finding optimal pairs of cooperative and competing patterns with bounded distance, in: Proceedings of the 7th International Conference on Discovery Science (DS 2004), in: Lecture Notes in Computer Science, vol. 3245, Springer-Verlag, 2004, pp. 32–46.
[20] S. Inenaga, H. Bannai, A. Shinohara, M. Takeda, S. Arikawa, Discovering best variable-length-don't-care patterns, in: Proceedings of the 5th International Conference on Discovery Science (DS 2002), in: Lecture Notes in Computer Science, vol. 2534, Springer-Verlag, 2002, pp. 86–97.
[21] S. Inenaga, T. Kivioja, V. Mäkinen, Finding missing patterns, in: Proceedings of the 4th Workshop on Algorithms in Bioinformatics (WABI 2004), in: Lecture Notes in Computer Science, vol. 3240, Springer-Verlag, 2004, pp. 463–474.
[22] J. Kärkkäinen, E. Ukkonen, Sparse suffix trees, in: Proceedings of the 2nd Annual International Computing and Combinatorics Conference (COCOON 1996), in: Lecture Notes in Computer Science, vol. 1090, Springer-Verlag, 1996, pp. 219–230.
[23] R. Karp, M. Rabin, Efficient randomized pattern-matching algorithms, IBM Journal of Research and Development 31 (1987) 249–260.
[24] J. Lanctot, M. Li, B. Ma, S. Wang, L. Zhang, Distinguishing string selection problems, Information and Computation 185 (1) (2003) 41–55.
[25] S.C. Li, Faster algorithms for finding missing patterns, in: Proceedings of the 12th Computing: The Australasian Theory Symposium (CATS 2006), 2006, pp. 107–111.
[26] X. Liu, D. Brutlag, J. Liu, BioProspector, discovering conserved DNA motifs in upstream regulatory regions of co-expressed genes, in: Proceedings of the Pacific Symposium on Biocomputing 2001, 2001, pp. 127–138.
[27] H. Mannila, H. Toivonen, A.I. Verkamo, Discovering frequent episodes in sequences, in: Proceedings of the 1st International Conference on Knowledge Discovery and Data Mining (KDD 1995), AAAI Press, 1995, pp. 210–215.
[28] L. Marsan, M.-F. Sagot, Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification, Journal of Computational Biology 7 (2000) 345–360.

[29] O.V. Matveeva, S.A. Shabalina, V.A. Nemtsov, A.D. Tsodikov, R.F. Gesteland, J.F. Atkins, Thermodynamic calculations and statistical correlations for oligo-probes design, Nucleic Acids Research 31 (14) (2003) 4211–4217.
[30] L. Parida, Pattern Discovery in Bioinformatics: Theory and Algorithms, Chapman & Hall, 2007.
[31] S. Shimozono, A. Shinohara, T. Shinohara, S. Miyano, S. Kuhara, S. Arikawa, Knowledge acquisition from amino acid sequences by machine learning system BONSAI, Transactions of Information Processing Society of Japan 35 (10) (1994) 2009–2018.
[32] A. Shinohara, M. Takeda, S. Arikawa, M. Hirao, H. Hoshino, S. Inenaga, Finding best patterns practically, in: Progress in Discovery Science, in: Lecture Notes in Artificial Intelligence, vol. 2281, Springer-Verlag, 2002, pp. 307–317.
[33] M. Takeda, S. Inenaga, H. Bannai, A. Shinohara, S. Arikawa, Discovering most classificatory patterns for very expressive pattern classes, in: Proceedings of the 6th International Conference on Discovery Science (DS 2003), in: Lecture Notes in Computer Science, vol. 2843, Springer-Verlag, 2003, pp. 486–493.
[34] J. Wang, B. Shapiro, D. Shasha, Pattern Discovery in Biomolecular Data, Oxford University Press, 1999.
[35] P. Weiner, Linear pattern matching algorithms, in: Proceedings of the IEEE 14th Annual Symposium on Switching and Automata Theory, 1973, pp. 1–11.